Pytest

Setup & Prerequisites

Important Python Keywords

The following keywords are a part of Python, and are not pytest-exclusive
They are often used when writing tests using pytest

pass

USE-CASE 1:
Write zero-logic tests that succeed every time

```
def test_demo():
  pass
```

```
collected 1 item

test_main.py .                                                    [100%]

============================ 1 passed in 0.01s =============================
```

USE-CASE 2: (Portal: JUMP99124)
Perform certain operations on the thing you're testing, which may or may not cause exceptions to be thrown
If your test hasn't crashed past a certain point, just pass the test

assert

The `assert` keyword can do 1 of 2 things depending on the expression placed after it

1. Passes the test if the expression resolves to True
2. Throws an AssertionError if the expression does not resolve to True

Example in regular python:

```python
x = 5
y = 10
assert x + y == 200
```

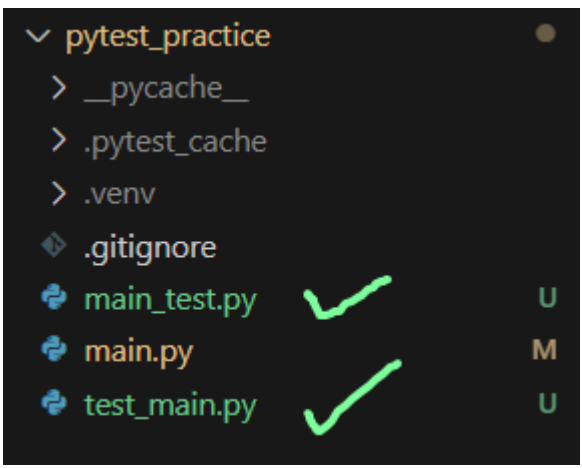
```
Traceback (most recent call last):
  File "c:\Users\owner\Desktop\playgrounds\pytest_practice\main.py", line 32, in <module>
    assert x + y == 200, "The sum of x and y should be 15"
           ^^^^^^^^^^^^
AssertionError: The sum of x and y should be 15
```

Setting up & Running Tests

Step 1: Create Test File

Once pytest is installed, you'll need to create files dedicated to running tests
To turn a python file into a test file, you have 2 options

1. Use `test_` as a prefix
2. Use `_test` as a suffix



Step 2: Create Test Functions

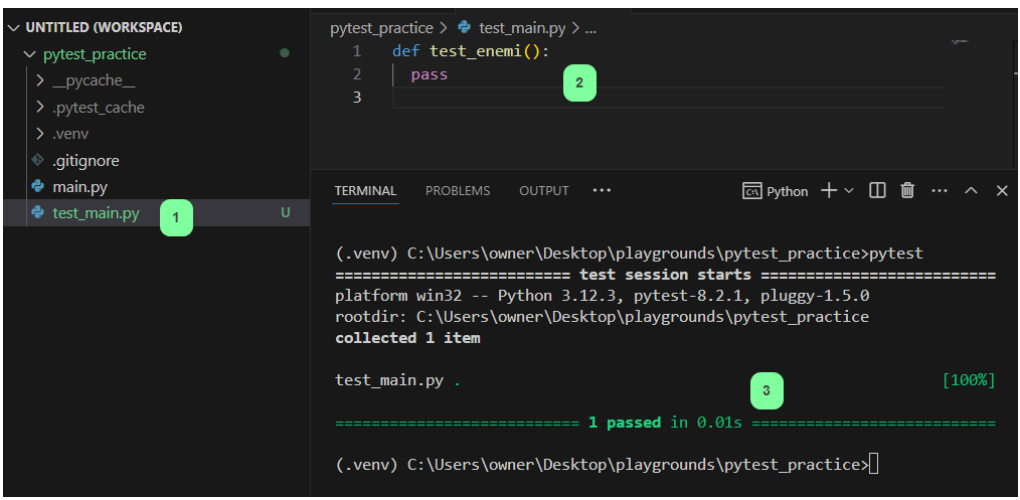
Every test is just a python function that either passes or fails
Prefix the function name with `test_`, or else Pytest won't be able to recognize it

```
def test_enemi():
  pass
```

Step 3: Run Tests

1. Open your terminal
2. Enter a directory where certain test files are located
3. Run the following command to execute all tests in the current directory

```
pytest
```



Good Practice for Test File Names

Lets say you have a file called `calcs.py`
If you're writing unit tests for that file, name it `test_calcs.py`

- Having `test_` at the start makes it easy to recognize what the test file is for
- Having the test file's name include the base file name makes it clear that `test_calcs.py` is a test file for `calcs.py` specifically

Run Only Specific Tests

Why this Matters

COMMON SCENARIO

- Professional codebases have 100s of test files
- Each test file may have multiple tests inside
- Entering `pytest` in the terminal will cause the entire test suite to run. If you are working on only 1 test at the moment, this could slow you down massively

SOLUTION

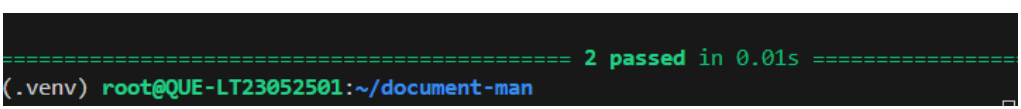
1. Use a special command to run `pytest` on 1 file specifically
2. Use a special command to run `pytest` on 1 test method inside 1 file specifically

Run all Tests in 1 File

Run all tests inside test_main.py

```
pytest test_main.py
```

OUTCOME
All 2 tests in test_main.py passed, while the entire test suite contains 3 tests total

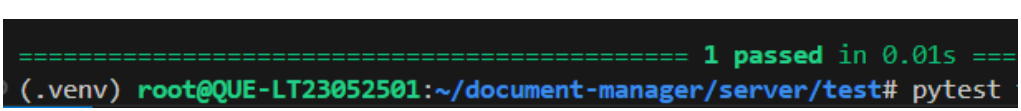


Run 1 Test in 1 File

Run 1 test inside test_main.py - specifically the `test_add_item_to_cart` method defined inside

```
pytest test_main.py::test_add_item_to_cart
```

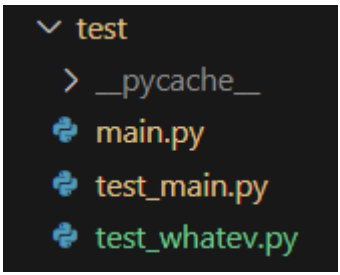
OUTCOME:
1 test in test_main.py passed, while the entire test suite contains 3 tests total



TEST_FILES

We have 2 tests in this file

test_main.py (2 tests inside) test_whatev.py (1 test inside)




```
def target(suffix: str):
    return
f"optchain_document_manager_server.routers.services.firebase_services.
{suffix}"
```

Enable print() during Tests

By default, any `print()` statements inside your test methods will not be shown in the terminal when the tests are ran
You can change this by using the `-s` flag - which is very useful for debugging

Run test without logs enabled:

```
pytest test_main.py::test_add_item_to_cart
```

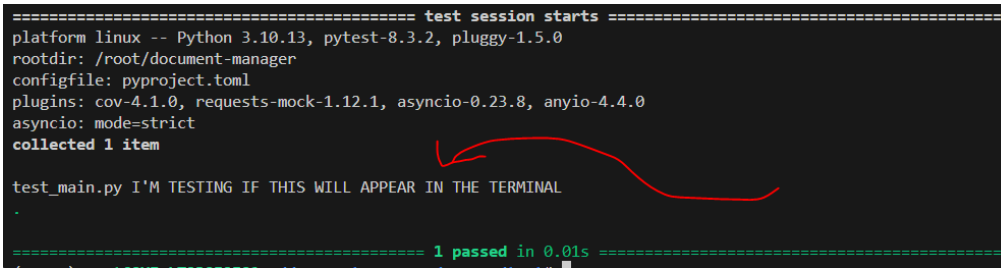
```
========================================= test session starts =========================================
platform linux -- Python 3.10.13, pytest-8.3.2, pluggy-1.5.0
rootdir: /root/document-manager
configfile: pyproject.toml
plugins: cov-4.1.0, requests-mock-1.12.1, asyncio-0.23.8, anyio-4.4.0
asyncio: mode=strict
collected 1 item

test_main.py .                                                                          [100%]

========================================== 1 passed in 0.01s ==========================================
```

Run test with logs enabled:

```
pytest test_main.py::test_add_item_to_cart -s
```



test_main.py

```
import pytest
def test_add_item_to_cart():
    print("I'M TESTING IF THIS WILL APPEAR IN THE TERMINAL"
    pass
```

Fundamentals

Pass test only when Exception is Raised

Default Behavior when Exceptions are Thrown

Any kind of exception thrown or raised inside a test will cause that test to fail, by default

- In the example below, we divide by 0 which causes a ZeroDivisionError
- The test fails automatically because an exception was thrown

```
from main import divide_nums
def test_divide():
    divide_nums(9, 0)
    pass # if test makes it here, just pass it (but it won't
```

```
main.py:5: ZeroDivisionError
========================================= short test summary info =========================================
FAILED test_main.py::test_divide - ZeroDivisionError: division by zero
========================================== 1 failed in 0.07s ==========================================
```

Deliberately Passing a Test when Exceptions are Thrown

pytest.raises

- Makes a test pass ONLY IF an exception is thrown
  You can specify a type of exception you want to be thrown, or settle for any generic one
- Use Case:
  Deliberately pass bad inputs to a method that should cause an error of some sort

If you don't get an exception thrown, then you know you need to tighten up your code (and testing showed you why)

```
import pytest
from main import divide_nums

def test_divide_v1():
    # Test for the prescence of a zero division error speci
    with pytest.raises(ZeroDivisionError):
        divide_nums(9, 0)      # PASSES
def test_divide_v2():
  # Test for any kind of exception occurring
    with pytest.raises(Exception):
        divide_nums(9, 0)      # PASSES

def test_divide_v3():
  # Test for any kind of exception occurring
    with pytest.raises(Exception):
        divide_nums(5, 9)      # FAILS BECAUSE NO ERRORS OCC
```

[divide_nums]

```
def divide_nums(x, y):
    return x / y
```

Fixtures: Reduce test Boiler Plate

When writing multiple tests, oftentimes we end up writing duplicate code while setting things up

- Fixtures allow us to set things up once then share it across as many tests as you want
  Similar to `setup_method` in Class-based tests
- This becomes extremely beneficial when setup/initialization takes more than just a couple lines

Repetitive Setup Code Example

In the two tests below, we end up initializing ShoppingCart and adding items to it
If we write more tests for ShoppingCart, we'll end up doing the same thing over and over again

test_main.py

```
from main import ShoppingCart
import pytest
def test_add_item_to_cart():
    # SETUP CODE BEGINS
    cart = ShoppingCart(2)
    # SETUP CODE ENDS
    cart.add_item({"item_name": "gum", "price": 2})
```

```
    item_count = cart.size()
    assert item_count == 1
def test_max_cart_size():
    with pytest.raises(OverflowError):
        # SETUP CODE BEGINS
        cart = ShoppingCart(2)  # cart should have max of 1
        # SETUP CODE ENDS
        cart.add_item({"item_name": "gum", "price": 2})
        cart.add_item({"item_name": "PS5", "price": 700})
        cart.add_item({"item_name": "COD MW3", "price": 90}
        pass
```

Solution: Fixtures

Let's reduce some of the setup code from the previous subsection

- Notice how we don't need to write `cart = ShoppingCart(2)` inside every test anymore
  We define our fixture `cart`, then have access to it as argument 1 of each test

- Keep in mind that fixtures get redefined each time a test is run

```
from main import ShoppingCart
import pytest
@pytest.fixture
def cart_with_max_2():
```

```
    return ShoppingCart(2)
def test_add_item_to_cart(cart_with_max_2):
    cart_with_max_2.add_item({"item_name": "gum", "price":
    item_count = cart_with_max_2.size()
    assert item_count == 1
def test_max_cart_size(cart_with_max_2):
    with pytest.raises(OverflowError):
        cart_with_max_2.add_item({"item_name": "gum", "pric
        cart_with_max_2.add_item({"item_name": "PS5", "pric
        cart_with_max_2.add_item({"item_name": "COD MW3", "
        pass
```

Works on both Test Types

fixtures in pytest can be used on class-based tests as well as method-based ones

With function-based tests:

```
pythonCopyimport pytest
@pytest.fixture
def example_fixture():
    return "fixture data"
def test_with_fixture(example_fixture):
    assert example_fixture == "fixture data"
```

With class-based tests:

```
pythonCopyimport pytest
@pytest.fixture
```

```
def example_fixture():
    return "fixture data"
class TestExample:
    def test_with_fixture(self, example_fixture):
        assert example_fixture == "fixture data"
```

[ShoppingCart Class]

main.py

```
from typing import List, TypedDict
class ItemDict(TypedDict):
    item_name: str
    price: float
class ShoppingCart:
    def __init__(self, max_items: int) -> None:
        self.items: List[ItemDict] = []
        self.max_items = max_items
    def add_item(self, item: ItemDict):
```

```
        if len(self.items) + 1 > self.max_items:
            raise OverflowError("Too many items added to ca
        self.items.append(item)
    def size(self) -> int:
        return len(self.items)
    def get_items(self) -> List[ItemDict]:
        return self.items
    def get_total_price(self):
        total = 0
        for item in self.items:
            total += item["price"]
        return total
```

Auto Use Fixtures for Class Methods

If you use `(autouse=True)` on a fixture that mocks a class-method

- It auto applies to every test
- You don't need to explicitly pass it to any of the tests
  To access the mock, simply call the method as you would normally (like it wasn't mocked at all)

main.py

```
import random
class WeatherService:
    def get_temperature(self):
        return random.randint(0, 30)  # Simulating an API c
    def get_weather_report(self):
        temp = self.get_temperature()
```

```
        if temp < 10:
            return "It's cold!"
        elif temp < 20:
            return "It's mild."
        else:
            return "It's hot!"
```

test_main.py

```
import pytest
from unittest.mock import patch
from main import WeatherService  # `GATE:8D051`
class TestWeatherService:
    @pytest.fixture(autouse=True)
    def setup(self):
        self.weather_service = WeatherService()
    @pytest.fixture(autouse=True)
    def cold_temperature(self):
        with patch.object(WeatherService, "get_temperature"
            yield
    def test_get_temp(self): # don't need to pass a fixture
        temp = self.weather_service.get_temperature()
        assert temp == 5

    def test_get_weather_report_cold(self):
```

```
        report = self.weather_service.get_weather_report()
        # SEE HOW get_temperature() results in 5 due to our
        assert report == "It's cold!"
```

Looped Mock Calls

Sometimes you will make requests to an API multiple times - and this can affect what tests we want to write

- How many times was a certain action performed?
- What arguments were used each time this action happened?

```
# Update the metadata of all associated entitities' docs
for doc in associated_docs:
    doc_data = doc.to_dict()
    doc_ref = self.firestore_client.collection(self.tenant_id).document(doc.id)

    if file_size is None:
        sizes_list = doc_data.get("file_sizes_mb", [0])
        file_size = sizes_list[0]
```

Remember that we mock any expensive actions that are taken in our tests

- In the application, perhaps we make a call to an API n times
- In our tests, we'll call a mocked method that simulates an API call n times

Count the Number of Mocked Method Calls

Check how many times a mocked method was called

`mock_name.call_count`

```python
def test_update_with_file(
    self,
    mock_set_doc_ref_data,
    mock_current_timestamp,
    payload,
    get_single_doc_result,
    get_associated_docs_result,
):
    with patch.object(DocumentPersistance, "get_single_doc", return_value=get_single_doc_result), patch.object(
        DocumentPersistance,
        "get_associated_docs",
        return_value=get_associated_docs_result,
    ):
        payload_model_instance = UpdateModel(**payload)
        file_format = "application/docx"
        file_size = 4.21
        self.persistence.update_metadata(payload_model_instance, True, file_size, file_format)

        args = mock_set_doc_ref_data.call_args
        assert mock_set_doc_ref_data.call_count == len(get_associated_docs_result)
```


See what Arguments were used per Call

Check which arguments are fed to a mocked call on each iteration

```python
for i, call in enumerate(mock_name.call_args_list):
 args, kwargs = call
```

```python
def test_update_with_file(
    self,
    mock_set_doc_ref_data,
    mock_current_timestamp,
    payload,
    get_single_doc_result,
    get_associated_docs_result,
):
    with patch.object(DocumentPersistance, "get_single_doc", return_value=get_single_doc_result), patch.object(
        DocumentPersistance,
        "get_associated_docs",
        return_value=get_associated_docs_result,
    ):
        payload_model_instance = UpdateModel(**payload)
        file_format = "application/docx"
        file_size = 4.21
        self.persistence.update_metadata(payload_model_instance, True, file_size, file_format)

        args = mock_set_doc_ref_data.call_args


        assert mock_set_doc_ref_data.call_count == len(get_associated_docs_result)

        for i, call in enumerate(mock_set_doc_ref_data.call_args_list):
            args, kwargs = call
```

Fixture with yield >

The difference between using `return` and `yield` in pytest fixtures is subtle but important. Let's break it down:

Using `return`:

1. When a fixture uses `return`, it executes the setup code and then immediately returns the value.
2. After the test function finishes, there's no way for the fixture to run any teardown code.
3. The fixture runs once and its returned value is cached for the scope of the fixture (function, class, module, etc.).

Using `yield`:

1. When a fixture uses `yield`, it executes the setup code, then pauses at the `yield` statement.
2. After the test function finishes, execution resumes in the fixture function right after the `yield` statement, allowing for teardown code.
3. The fixture still runs once per scope and its value is cached, but it allows for both setup and teardown phases.

We chose `yield` in this case for several reasons:

1. **Resource Management**: Even though we don't have explicit teardown code after the `yield`, using `yield` ensures that the context manager (the `with` statement) properly closes and cleans up the patched objects after each test. This is crucial for maintaining a clean test environment.
2. **Flexibility**: If we later need to add teardown code (e.g., clearing a database, closing connections), we can easily add it after the `yield` without changing the fixture's interface.
3. **Clarity**: Using `yield` clearly separates the setup phase from any potential teardown phase, making the fixture's structure more readable and maintainable.
4. **Consistency with Complex Fixtures**: While this fixture is relatively simple, using `yield` maintains consistency with more complex fixtures that might require teardown. This makes the codebase more uniform and easier to understand.
5. **Context Preservation**: In more complex scenarios, `yield` allows the fixture to maintain its execution context, which can be useful for stateful fixtures or those that need to perform actions based on how the test executed.

```python
@pytest.fixture
def fixture_with_return():
    # Setup
    resource = setup_resource()
    return resource
    # No opportunity for teardown
@pytest.fixture
def fixture_with_yield():
    # Setup
    resource = setup_resource()
    yield resource
    # Teardown
    cleanup_resource(resource)
```

Marks

Overview

Pytest marks allow you to add metadata to your tests
This can be used for lots of things like skipping tests, allowing them to fail, allowing you to test them in named groups...etc

Built in Marks

These marks are built into Pytest by default

Custom Marks

UNOFFICIAL EXAMPLES:
These are unofficial marks, and you can create new ones for other purposes later

You can create your own marks by using `@pytest.mark.custom_mark_name`

Run Marked Groups

Use a special command to only run tests with specific marks
This pairs well with custom marks since it allows you to split your tests into groups

```
pytest -m custom_mark_name
```

Built-in Marks

xfail

Allows your test to fail without crashing your test suite

- If an xfail marked test does fail, it is considered `xfailed`
- If an xfail-marked test succeeds, it is considered `xpassed`

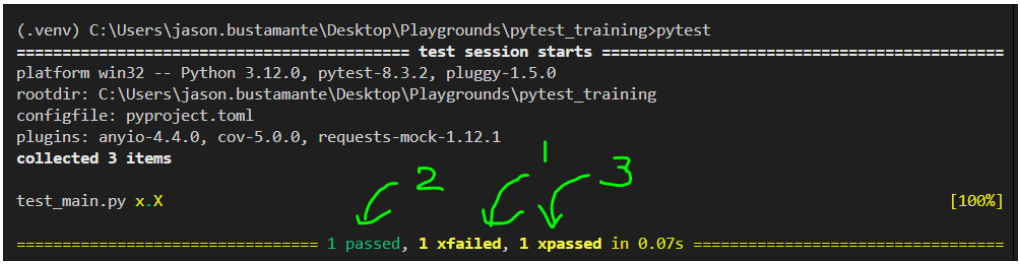
main.py

```
def divide_nums(x, y):
    return x / y
```

test_main.py

```
from main import divide_nums
import pytest
@pytest.mark.xfail(reason="Dividing by zero results in an e
def test_divide_v1():
    divide_nums(9, 0) # xfailed
    pass
def test_divide_v2():
    result = divide_nums(9, 1) # passed
    pass

@pytest.mark.xfail(reason="Dividing by zero results in an e
def test_divide_v3():
    result = divide_nums(9, 1) # xpassed
    pass
```



skip

Skips the test completely

main.py

```
def divide_nums(x, y):
    return x / y
```

test_main.py

```
from main import divide_nums
import pytest
@pytest.mark.skip(reason="Skip since this test will always
def test_divide_v1():
    divide_nums(9, 0)
    pass
def test_divide_v2():
    result = divide_nums(9, 1) # passed
    pass
```

```
(.venv) C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training>pytest
========================================= test session starts =========================================
platform win32 -- Python 3.12.0, pytest-8.3.2, pluggy-1.5.0
rootdir: C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training
configfile: pyproject.toml
plugins: anyio-4.4.0, cov-5.0.0, requests-mock-1.12.1
collected 2 items

test_main.py s.                                                                              [100%]

=================================== 1 passed, 1 skipped in 0.01s ====================================
```

Custom Marks

Quick Summary

Use custom marks to group your tests
You then have the ability to run your groups with the following command

```
pytest -m custom_mark_name
```

Demonstration

main.py

```
def divide_nums(x, y):
    return x / y
```

test_main.py

```
from main import divide_nums
import pytest
```

```python
@pytest.mark.edge
def test_divide_v1():
    divide_nums(0, 1) # xfailed
    pass
@pytest.mark.normal
def test_divide_v3():
    result = divide_nums(8, 4) # passed
    pass
@pytest.mark.xfail
def test_divide_v4():
    result = divide_nums(9, 0)
    pass
```

command: `pytest`

```
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============================== 2 passed, 1 xfailed, 2 warnings in 0.07s ==============================
```

command: `pytest -m normal`

```
    @pytest.mark.normal

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============================== 1 passed, 2 deselected, 2 warnings in 0.01s ==============================
```

Parametrize Mark: Pass Different Sets of Params

The parametrize mark allows you to run a test multiple times - with different parameters fed to the class or method you're testing
Saves you from having to re-write a lot of boiler plate code

Syntax

WITHOUT PARAMETERIZATION:
Must write 3 test functions to test 3 sets of inputs

main.py

```python
def divide_nums(x, y):
    return x / y
```

test_main.py (NO PARAMETRIZATION)

```
def test_divide_v1():
    assert divide_nums(0, 1) == 0
def test_divide_v2():
    assert divide_nums(8, 4) == 2
def test_divide_v3():
    assert divide_nums(9, 1) == 1
```

test_main.py (WITH)

```
import pytest
def divide_nums(x, y):
    return x / y # METHOD WE ARE TESTING
@pytest.mark.parametrize("x, y, quotient", [(0, 1, 0), (8,
def test_divide(x, y, quotient):
    assert divide_nums(x, y) == quotient
```

= = = = = = = = = TEST TYPES

= = = = = = = = =

Method-based

Test a Simple Function

GENERAL GAMEPLAN

- Import the method
- Call the method
- Test the method's behavior with certain parameters provided

main.py

```
def add_nums(x, y):
    return x + y
```

test_main.py

```
from main import add_nums
def test_add_nums():
    assert add_nums(1, 5) == 6 # TEST PASSES
```

Test a Simple Class

GENERAL GAMEPLAN:

- Import the class
- Initialize a class instance

- Test the behavior of class methods

TEST PURPOSE:
Test if the add_item method increases the length of the items list

main.py

```
from typing import List, TypedDict
class ItemDict(TypedDict):
    item_name: str
    price: float
class ShoppingCart:
    def __init__(self, max_size: int) -> None:
        self.items: List[ItemDict] = []
    def add_item(self, item: ItemDict):
        self.items.append(item)
    def size(self) -> int:
        return len(self.items)
    def get_items(self) -> List[ItemDict]:
        return self.items
    def get_total_price(self):
```

```
        total = 0
        for item in self.items:
            total += item["price"]
        return total
```

test_main.py

```
from main import ShoppingCart
def test_add_item_to_cart():
    # cart item count should increase by 1 when you add an
    cart = ShoppingCart(5)
    cart.add_item({ "item_name": "banana", "price": 0.5 })
    item_count = cart.size()
    assert item_count == 1
```

```
collected 1 item

test_main.py .

========================================================= 1 passed in 0.02s =
```

Class-based

Setup + Teardown Methods

Overview

Good example of using a setup method in the next lesson
(teleport: j139)

Demo

```
class TestDemo:
    def setup_method(self, method):
        print(f"===== Setting up {method} =====")
    def teardown_method(self, method):
        print(f"===== Tearing down {method} =====")
    # Empty test that does nothing but print a line
    def test_one(self):
        print("TEST METHOD PRINT STATEMENT")
        pass
```

```
test_main.py ===== Setting up <bound method TestCircle.test_one of <test_main.TestCircle object at 0x000001CBE7DC9520>> =====
TEST METHOD PRINT STATEMENT
.===== Tearing down <bound method TestCircle.test_one of <test_main.TestCircle object at 0x000001CBE7DC9520>> =====


============================================================ 1 passed in 0.02s ============================================================
```

Setup Method: 2 Variants

There are two ways we can name our setup methods while writing class-based tests

setUp

Set calc.mode to "float" for every test

```
class TestCalculator:
    def setUp(self):
        self.calc = Calculator()

        # Change calc.mode based on which test is being ran
        self.calc.mode = "float"
    def test_addition(self):
        assert self.calc.add(2, 3) == 5
    def test_division(self):
        assert self.calc.divide(5, 2) == 2.5
```

setup_method

- Set calc.mode to "float" for the test_division
- Set calc.mode to "integer" for the test_addition

```
class TestCalculator:
    def setup_method(self, method):
        self.calc = Calculator()

        # Change calc.mode based on which test is being ran
        if method.__name__ == "test_division":
            self.calc.mode = "float"
        else:
            self.calc.mode = "integer"
    def test_addition(self):
        assert self.calc.add(2, 3) == 5
    def test_division(self):
        assert self.calc.divide(5, 2) == 2.5
```

Test a Simple Function

main.py

```
def rectangle(length, width):
    return {"area": length * width, "perimeter": (length *
```

test_main.py

```
import pytest
from main import rectangle
class TestTwoByFiveRectangle:
    def setup_method(self):
        # Initialize the class using a setup method
        self.rect = rectangle(2, 5)
    def test_area(self):
        area = self.rect["area"]
        assert area == 10

    def test_perimeter(self):
        area = self.rect["perimeter"]
        assert area == 14
```

Test a Simple Class

main.py

```
class Rectangle():
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
    def perimeter(self):
        return (self.length * 2) + (self.width * 2)
```

test_main.py

```python
import pytest
from main import Rectangle
class TestTwoByFiveRectangle:
    def setup_method(self):
        # Initialize the class using a setup method (telepo
        self.rect = Rectangle(2, 5)
    def test_area(self):
        area = self.rect.area()
        assert area == 10

    def test_perimeter(self):
        area = self.rect.perimeter()
        assert area == 14
```

= = = = = = = = MOCKING = = = = = = = =

Intro to Mocking

What should I Mock?

Complex software is made of several moving parts - Rest API's, databases, Cloud Services, and more
When writing unit tests, you should test the logic of the code YOU wrote - not whether these 3rd party services work

Mock 3rd Party Services

Mocking allows you to replace 3rd party dependencies with mock objects that simulate their behavior

- Replace everything outside of your code's control with mocks
- Test what the logic of the code you wrote

EXAMPLE:

- Mock the parts where you save things to firestore and cloud storage (highlighted yellow)
- Test that the metadata object you generate has a name and size field

```
def save_file(uid, file):
    metadata = {
        "name": file.name,
        "size": file.size
    }
    firestore.collection("metadata").document(uid).set(metadata)
    cloud_storage.blob(file).save()

    return metadata
```

logic

Bad Practice: Making API Calls during Tests

There are many reasons why making API calls during tests is a bad idea

- Slows down test execution time
- Racks up expenses since most 3rd party services like Google Cloud aren't entirely free
- The 3rd party dependencies may fail outside of your control
  Service outage, errors in latest patch, bad connection when making a request to them... etc

The 3 Mocking Variants

We have 3 main ways to mock 3rd party dependencies
They accomplish the same things but differ in terms of syntax

1. Decorator Mocking
2. Context Manager Mocking
3. Inline Mocking

Pytest vs unittest

Python actually has a built-in test library called `unittest` - it's an OOP-based testing framework

- You can set up and write your tests with it
- Has excellent tools for mocking 3rd party dependencies

Pytest is a 3rd party testing framework that's functional and much simpler to learn

- In this Typora file, we're setting up our tests using Pytest
- We still use `unittest.mock` for mocking 3rd party dependencies

  If you want to compare Pytest vs unittest, read the following article:
  https://realpython.com/pytest-python-testing/

mock.patch()

Syntax

mock a variable value

```
@patch("main.FACTOR", 10)
```

mock a method

```
@patch("main.some_method", return_value="1a1df")
```

String Parameter

mock.patch requires only 1 positional argument - the path to the method you plan to mock

- The path is relative to the directory where the test command is run
- This is usually the project root, so in practice you should treat this path like an absolute one

Keyword Arg: return_value

Allows you specify the return value of a method that you're mocking

Method returns a string

```
mock.patch('package.fileName.methodName', return_value="Function
output")
```

Method returns a dict with detail kvp

```
mock.patch('package.fileName.methodName', return_value={ 'detail':
'success' })
```

Keyword arg: side_effect >

Idk what this does

mock.patch.object()

This method is used to mock specific attributes of a class object

Syntax + Purpose

```
from unittest.mock import patch
@patch.object(target, attribute, new=DEFAULT, ...)
```

target:
The object that contains the attribute you want to mock

attribute:
the string that points to the attribute you wish to mock

new:
The new value you plan to temporarily replace the old one with
If you leave it unspecified, `MagicMock()` will be used as the default value

Common Use Case: Mocking Class Methods

The patch.object method is most commonly used for mocking the return value of methods defined in a class
When a class is initialized, it becomes an object whose methods are considered attributes on it

main.py

```
import random
class WeatherService:
    def get_temperature(self):
        return random.randint(0, 30)  # Simulating an API c
    def get_weather_report(self):
        temp = self.get_temperature()
        if temp < 10:
            return "It's cold!"
        elif temp < 20:
```

```
            return "It's mild."
        else:
            return "It's hot!"
```

test_main.py

```
import unittest
import pytest
from unittest.mock import patch, Mock
from main import WeatherService # `GATE:8D051`
class TestWeatherService(unittest.TestCase):
    def setUp(self):
        # Initialize class
        self.weather_service = WeatherService()
    # patch.object Context Manager
    def test_get_weather_report_cold_1(self):
        with patch.object(self.weather_service, "get_temper
            report = self.weather_service.get_weather_repor
            assert report == "It's cold!"
    # patch.object decorator
    @patch.object(WeatherService, "get_temperature", return
    def test_get_weather_report_hot(self, mocked_get_temper
        self.weather_service = WeatherService()
        report = self.weather_service.get_weather_report()
```

```
        assert report == "It's hot!"
```

Use Fixtures

Feel free to make these mocks into fixtures

```
import pytest
from unittest.mock import patch
from main import WeatherService  # `GATE:8D051`
class TestWeatherService:
    @pytest.fixture(autouse=True)
    def setup(self):
        self.weather_service = WeatherService()
    @pytest.fixture
    def cold_temperature(self):
        with patch.object(WeatherService, "get_temperature"
            yield
    @pytest.fixture
    def hot_temperature(self):
        with patch.object(WeatherService, "get_temperature"
            yield
```

```
    def test_get_weather_report_cold(self, cold_temperature
        report = self.weather_service.get_weather_report()
        assert report == "It's cold!"
    def test_get_weather_report_hot(self, hot_temperature):
        report = self.weather_service.get_weather_report()
        assert report == "It's hot!"
```

mock.patch.dict()

This method allows you to temporarily modify dicts or environment variables during testing
We can edit existing KVPs or add new ones

main.py

```
import random
config = {
    "multiplier": {
        "factor": 5,
    },
}
def amplify(num):
    return num * config["multiplier"]["factor"]
```

test_main.py

```
import unittest
import pytest
from unittest.mock import patch, Mock
from main import amplify, config # JUMP:J43ADF
def test_amplify_context_manager():
    with patch.dict(config["multiplier"], {"factor": 10}):
        assert amplify(5) == 50
@patch.dict(config["multiplier"], {"factor": 10})
def test_amplify_decorator():
    assert amplify(5) == 50
```

side_effect Param for patch() Methods

Every `patch()` method in unittest has an optional parameter called side_effect

Most Common Use Case

The most common use case for `side_effect` is to set it equal to an exception It forces that exception to be raised every time the mock is called

main.py

```
import requests
def get_user_data(post_id):
    try:
        response = requests.get(f"https://jsonplaceholder.t
        response.raise_for_status() # raise exception if 40
        return response.json()
    except requests.exceptions.RequestException as e:
        raise RuntimeError(f"Failed to fetch user data: {st
```

test_main.py

```
import unittest
import pytest
import requests
from unittest.mock import patch, Mock
from main import get_user_data
def test_get_user_data_api_error():
    # Create a mock with a side effect, then patch request.
    # Side effect is an exception - one specifically from t
    mock_requests_get = Mock(side_effect=requests.Connectio

    with patch("requests.get", mock_requests_get):
        # Assert that calling get_user_data raises RuntimeE
        with pytest.raises(RuntimeError) as exc_info:
            get_user_data(123)
            error_msg = str(exc_info.value)
        assert "Failed to fetch user data: API is down" in
```

Other Use Cases >

https://docs.python.org/3/library/unittest.mock.html#:~:text=an%20AttributeError.-,side_effect,-%3A%20A%20function%20to

https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock.side_effect

Mocking 3rd Party Dependencies + The 3 Mocking Variants

Option 1: Decorator Mocking

Attach a decorator on top of your test_methods

Demo

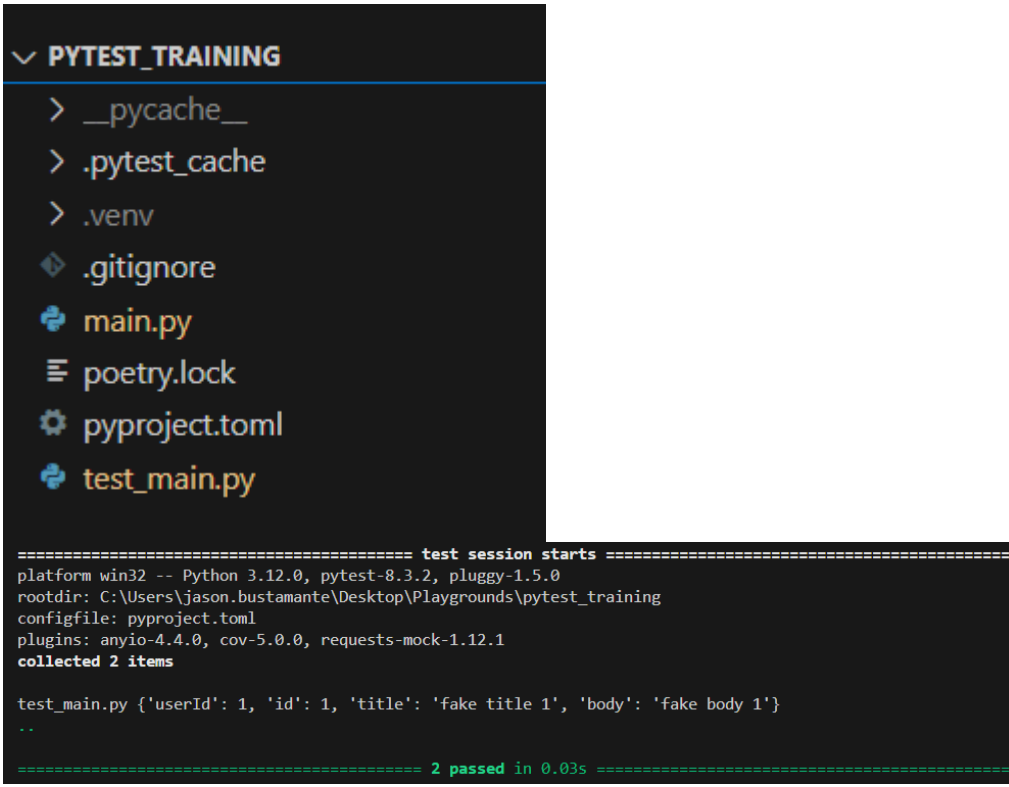
main.py

```
import requests
def get_user_data(user_id):
    response = requests.get(f"https://jsonplaceholder.typic
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception
```

test_main.py

```python
import unittest
import pytest
from unittest.mock import patch, Mock
from main import get_user_data
@patch("main.requests.get")
def test_get(mocked_request_get):
    # Configure a mock object and set it equal to the mocke
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {
        "userId": 1,
        "id": 1,
        "title": "fake title 1",
        "body": "fake body 1",
    }
    mocked_request_get.return_value = mock_response
    # Call the imported method you are testing (the request
    data = get_user_data(1)
    print(data)
    assert data["userId"] == 1
@patch("main.requests.get")
```

```python
def test_failed_get(mocked_request_get):
    mock_response = Mock()
    mock_response.status_code = 404
    # We skip out on assigning a json return value since st
    mocked_request_get.return_value = mock_response
    with pytest.raises(Exception):
        data = get_user_data(1)
        pass
```



Visual Aid

mocked_request_get

- Is the parameter in each of our tests methods
- Refers to the mock function we created using `@patch("main.requests.get")`

```python

@patch("main.requests.get")
def test_get(mocked_request_get):
    # Configure a mock object and set it equal to the mocked method's return value
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {
        "userId": 1,
        "id": 1,
        "title": "fake title 1",
        "body": "fake body 1",
    }
    mocked_request_get.return_value = mock_response

    # Call the imported method you are testing (the request.get method will now be mocked
    data = get_user_data(1)
    print(data)
    assert data["userId"] == 1

```

LINE BY LINE EXPLANATION

1. Initialize a mock object
2. Set the status_code property of the mock object
3. set the return value of the json() method of this mock object
    - Keep in mind that everything in Python is an object
    - If we were mocking firestore.collection(), we'd use `mock_response.collection.return_value = something`
6. Set the return value for the mocked method named `mocked_request_get`

Option 2: Context Manager Mocking

This variant involves creating a new scope inside your test method

Example

main.py

```
import requests
def get_user_data(user_id):
    response = requests.get(f"http://localhost:8000/{user_i
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception
```

test_main.py

```
import unittest
import pytest
```

```python
from unittest.mock import patch, Mock
from main import get_user_data
def test_get():
    with patch("main.requests.get") as mocked_request_get:
        # Configure mock object and assign it to mocked met
        mock_response = Mock()
        mock_response.status_code = 200
        mock_response.json.return_value = {
            "userId": 1,
            "id": 1,
            "title": "fake title 1",
            "body": "fake body 1",
        }
        mocked_request_get.return_value = mock_response
        # Call the method we're testing
        data = get_user_data(1)
        print(data)
        assert data["userId"] == 1
def test_failed_get():
    with patch("main.requests.get") as mocked_request_get:
        mock_response = Mock()
        mock_response.status_code = 404
        # We skip out on assigning a json return value sinc
        mocked_request_get.return_value = mock_response
        with pytest.raises(Exception):
            data = get_user_data(1)
            pass
```

```
========================================= test session starts =========================================
platform win32 -- Python 3.12.0, pytest-8.3.2, pluggy-1.5.0
rootdir: C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training
configfile: pyproject.toml
plugins: anyio-4.4.0, cov-5.0.0, requests-mock-1.12.1
collected 2 items

test_main.py {'userId': 1, 'id': 1, 'title': 'fake title 1', 'body': 'fake body 1'}
..

========================================== 2 passed in 0.04s ==========================================
```

Syntax Variations

Seems you can configure a `Mock` or `MagicMock` option before your patch, then assign it as its value
This is different than the previous subsection's approach where we patch first then configure later

```python
import pytest
from unittest.mock import Mock, patch
import requests
def get_user_data(user_id):
    try:
        response = requests.get(f"https://api.example.com/u
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        raise RuntimeError(f"Failed to fetch user data: {st
def test_get_user_data_api_error():
    # Create a mock with a side effect
    mock_requests_get = Mock(side_effect=requests.Connectio
    # Patch the requests.get function with our mock
    with patch('requests.get', mock_requests_get):
        # Assert that calling get_user_data raises RuntimeE
        with pytest.raises(RuntimeError) as exc_info:
            get_user_data(123)

    # Additional assertions
    assert "Failed to fetch user data: API is down" in str(
    mock_requests_get.assert_called_once_with("https://api.
```

OR

```
import pytest
from unittest.mock import Mock, patch
import requests
def get_user_data(user_id):
    try:
        response = requests.get(f"https://api.example.com/u
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        raise RuntimeError(f"Failed to fetch user data: {st
@patch('requests.get')
def test_get_user_data_api_error(mock_requests_get):
    # Configure the mock after patching
    mock_requests_get.side_effect = requests.ConnectionErro
    # Assert that calling get_user_data raises RuntimeError
    with pytest.raises(RuntimeError) as exc_info:
        get_user_data(123)

    # Additional assertions
    assert "Failed to fetch user data: API is down" in str(
    mock_requests_get.assert_called_once_with("https://api.
```

Option 3: Inline Mocking

Syntax

main.py

```
import requests
def get_user_data(user_id):
    response = requests.get(f"http://localhost:8000/{user_i
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception
```

test_main.py

```python
import unittest
import pytest
from unittest.mock import patch, Mock
from main import get_user_data
def test_get():
    # Configure mock object
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {
        "userId": 1,
        "id": 1,
        "title": "fake title 1",
        "body": "fake body 1",
    }
    # Create a mock method for requests.get which returns t
    mocked_request_get = Mock(return_value=mock_response)
    # Temporarily replace requests.get with mocked_request_
    with patch("main.requests.get", mocked_request_get):
        data = get_user_data(1)  # call method we're testin
        print(data)
        assert data["userId"] == 1
def test_failed_get():
    # Configure mock object
    mock_response = Mock()
    mock_response.status_code = 404
    # We skip out on assigning a json return value since st

    # Create a mock method for requests.get which returns t
```

```
    mocked_request_get = Mock(return_value=mock_response)

    # Temporarily replace requests.get with mocked_request_
    with patch("main.requests.get", mocked_request_get):
        with pytest.raises(Exception):
            data = get_user_data(1)
            pass
```

```
========================================= test session starts =========================================
platform win32 -- Python 3.12.0, pytest-8.3.2, pluggy-1.5.0
rootdir: C:\Users\jason.bustamante\Desktop\Playgrounds\pytest_training
configfile: pyproject.toml
plugins: anyio-4.4.0, cov-5.0.0, requests-mock-1.12.1
collected 2 items

test_main.py {'userId': 1, 'id': 1, 'title': 'fake title 1', 'body': 'fake body 1'}
..

========================================== 2 passed in 0.04s ==========================================
```

Use with Parameterized Marks

Inline mocking synchronizes well with parameterized marks.
You can use the different values from your parameterized mark datasets when defining your inline mock

```
@pytest.mark.parametrize(
    "payload, get_doc_data_result, get_associated_docs_
    [
        (
            UPDATE_PAYLOAD_1,
```

```
                METADATA_SAMPLE_1,
                [METADATA_SAMPLE_1, METADATA_SAMPLE_2, META
            )
        ],
    )
    def test_update_with_no_file(self, payload, get_doc_dat
        payload_model_instance = UpdateModel(**payload)
        self.persistence.update_metadata(payload_model_inst
        #
        with patch.object(DocumentPersistance, "get_doc_dat
            DocumentPersistance, "get_associated_docs", ret
        ):
            # assert something
            pass
```

Multiple Mocks Syntax

Showcase all 3 variants or tack them on the tail end of each lesson

```
import unittest
from unittest.mock import patch, Mock
# Assume we have a function that uses both a database and a
def get_user_data(user_id):
    db_data = database.get_user(user_id)
    api_data = requests.get(f"https://api.example.com/user/
    return {**db_data, **api_data.json()}
# ... add in either of the 3 code snippets to the end of th
```

Decorator

```
@patch("module.database.get_user")
@patch("module.requests.get")
def test_get_user_data_decorator(mock_requests_get, mock_db
    # Configure mocks
    mock_db_get_user.return_value = {"id": 1, "name": "John
    mock_api_response = Mock()
    mock_api_response.json.return_value = {"email": "john@e
    mock_requests_get.return_value = mock_api_response
    # Call the function
    result = get_user_data(1)
    # Assertions
    assert result == {"id": 1, "name": "John Doe", "email":
    mock_db_get_user.assert_called_once_with(1)
    mock_requests_get.assert_called_once_with("https://api.
```

Context Manager

```
def test_get_user_data_context_manager():
    with patch("module.database.get_user") as mock_db_get_u
         patch("module.requests.get") as mock_requests_get:
```

```
        # Configure mocks
        mock_db_get_user.return_value = {"id": 1, "name": "
        mock_api_response = Mock()
        mock_api_response.json.return_value = {"email": "jo
        mock_requests_get.return_value = mock_api_response
        # Call the function
        result = get_user_data(1)
        # Assertions
        assert result == {"id": 1, "name": "John Doe", "ema
        mock_db_get_user.assert_called_once_with(1)
        mock_requests_get.assert_called_once_with("https://
```

Inline

```
def test_get_user_data_inline():
    # Create mock objects
    mock_db_get_user = Mock(return_value={"id": 1, "name":
    mock_api_response = Mock()
    mock_api_response.json.return_value = {"email": "john@e
    mock_requests_get = Mock(return_value=mock_api_response
    # Apply mocks
    with patch("module.database.get_user", mock_db_get_user
         patch("module.requests.get", mock_requests_get):
        # Call the function
        result = get_user_data(1)
    # Assertions
    assert result == {"id": 1, "name": "John Doe", "email":
    mock_db_get_user.assert_called_once_with(1)
    mock_requests_get.assert_called_once_with("https://api.
```

Implicit Mocks: Shorthand

In previous examples we've explicitly declared mock objects, configured them, then set them equal to return value of a mocked function
However, Pytest offers shortcuts to make this easier

Demo

NOTICE:
We do not declare `Mock()` or specify return_value for anything

EXPLANATION:

- When used as a decorator `@patch`, replaces the path's target with a `MagicMock()` instance for the duration of the test
  This is why we do not see `Mock()` or `MagicMock` declared anywhere
- setup_method runs at the start of every test thanks to `autouse=True` , so test_silly gets access to `self.document_persistence`

```
import pytest
from unittest.mock import AsyncMock, MagicMock, patch
from fastapi import HTTPException, UploadFile
from document_manager_server.routers.constants.classes impo
```

```
from document_manager_server.routers.constants.firebase_ser
TENANT_ID = "905EB"
class TestCreate:
    @pytest.fixture(autouse=True)
    @patch("firebase_admin.storage.bucket")
    @patch("google.cloud.storage.Client")
    @patch("firebase_admin.firestore.client")
    def setup_method(self, mock_firestore_client, mock_stor
        # Initialize the DocumentPersistance class with moc
        # This methods runs before each test, meaning each
        self.document_persistence = DocumentPersistance(TEN
    def test_silly(self):
        pass
```

Longform Version

If we did not use implicit mocking, we'd need to assign Mock objects to the 3 methods one by one

```python
import pytest
from unittest.mock import MagicMock, patch
from chain_document_manager_server.routers.constants.fireba
TENANT_ID = "905EB"
class TestCreate:
    @pytest.fixture(autouse=True)
    def setup_method(self):
        # Create mock objects
        mock_bucket = MagicMock()
        mock_storage_client = MagicMock()
        mock_firestore_client = MagicMock()
        # Use patch as a context manager
        with patch("firebase_admin.storage.bucket", return_
             patch("google.cloud.storage.Client", return_va
             patch("firebase_admin.firestore.client", retur
            # Initialize DocumentPersistance with mocked se
            self.document_persistence = DocumentPersistance
            # Make mocks available as instance attributes f
            self.mock_bucket = mock_bucket
            self.mock_storage_client = mock_storage_client
            self.mock_firestore_client = mock_firestore_cli
            # Yield control back to pytest, which will run
            yield
        # The context manager will automatically stop the p
    def test_silly(self):
        pass
```

How to Mock Other Stuff

Variables: Defined Outside of Method Only

Which Variables can we Mock?

You cannot mock variables defined inside of methods using standard mocking techniques
You can mock variables defined outside of methods you are testing however

CANNOT MOCK:
Variables defined inside the method

CAN MOCK:

- Variables defined outside of the method
- Variables imported from external files

```
def amplify(num):
    FACTOR = 2
    return num * FACTOR
```

```
FACTOR = 2
def amplify(num):
    return num * FACTOR
```

```
from constants import FACTOR

def amplify(num):
    return num * FACTOR
```

How to Mock a Variable

EXAMPLE:

- Mock the `FACTOR` variable defined outside of the function we're testing
- The syntax would be the exact same if FACTOR was imported from an external file, instead of being defined outside the method

main.py

```
FACTOR = 2
def amplify(num):
    return num * FACTOR
```

test_main.py

```
import unittest
import pytest
from unittest.mock import patch, Mock
from main import amplify
def test_amplify_v1():
    with patch("main.FACTOR", 10) as mocked_multiplier:
        result = amplify(5)
        assert result == 50
@patch("main.FACTOR", 10)
def test_amplify_v2():
    result = amplify(5)
    assert result == 50
```

mock.patch() Syntax Re-explained

You may have noticed we use `@patch` without a return_value keyword argument

- This is because we directly assign a value to the variables we mock
- Typically return_value is used only when mocking functions or methods, which actually have values they return - while a variable does not

mock a variable

```
@patch("main.FACTOR", 10)
```

mock a method

```
@patch("main.generate_uuid", return_value="1a1df")
```

Mock a Specific Attribute on a Dict

If you want to mock an attribute in a dict, use `mock.patch.dict()`
You could mock the entire thing with `mock.patch()`, but that's a lot of extra code

`JUMP:J43ADF`

Methods

OBJECTIVE:
Mock the generate_random method we import to main.py from a different file
Keep in mind that mocking methods from 3rd party dependencies works the same way

Functional Methods

main.py

```
from util.methods import generate_random
def random_return():
    return generate_random()
```

methods.py

```
import random
def generate_random():
    return random.randint(1, 10)
```

test_main.py

```
import unittest
import pytest
from unittest.mock import patch, Mock
from main import random_return
@patch("main.generate_random", return_value=9)
def test_random_return_decorator(mocked_generate_random):
    result = random_return()
    assert result == 9
def test_random_return_context_manager():
    with patch("main.generate_random", return_value=9) as m
        result = random_return()
        assert result == 9
```

Notice how mocked_generate_random is not used anywhere in the test after we declare it
It's because our mocked method already has a return_value set, and there's nothing else we need to do with it

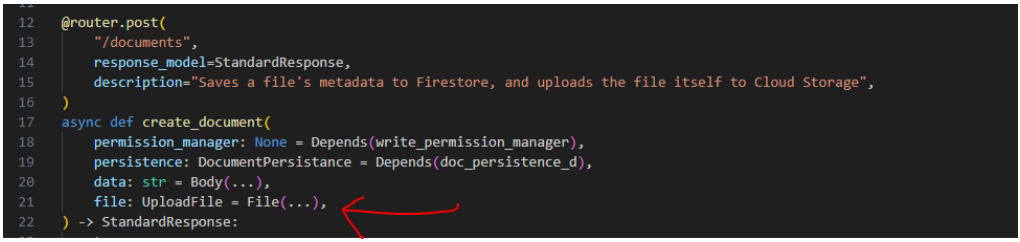
Methods Defined Inside a Class

When mocking class methods, use `mock.patch.object()`
See an example in `GATE:8D051`

Method Parameters (unsure if works)

Sometimes you pass certain things to methods that cannot be properly replicated with Python

Example: Mock an UploadFile



We send a PDF file to an endpoint in the request body
When testing the endpoint, we can use a configured Mock object to imitate that file

main.py (contains method we are testing)

```
    def playground(self, file: UploadFile):
        pass
```

test_file

```
import pytest
from unittest.mock import AsyncMock, MagicMock, patch
from fastapi import UploadFile
class TestPersistence:
    # Define a fixture that returns a mocked file
    @pytest.fixture
    def mock_upload_file(self):
        file = MagicMock(
            spec=UploadFile
        )  # ensures that the mock will have the same attri
        file.filename = "chiro_receipt"
        file.content_type = "application/pdf"
        file.file = MagicMock()
        return file

    # In your tests, pass in the fixture then use it when i
 def test_playground(self, mock_upload_file):
        self.persistence.playground(mock_upload_file)
        pass
```

Documentation

Pytest + unittest Tutorial
https://www.youtube.com/watch?v=cHYq1MRoyI0

unittest.mock.patch tutorial (EXCELLENT)
https://www.youtube.com/watch?v=WFRljVPHrkE

Mock Cheat Sheet
http://www.ines-panker.com/2020/06/01/python-mock.html

Advanced Assert Methods unittest provides
https://kapeli.com/cheat_sheets/Python_unittest_Assertions.docset/Contents/Resources/Documents/index

Documentation Links

Mock() vs MagicMock()
The `Mock` and `MagicMock` methods can oftentimes be used interchangeably, but MagicMock has a specialty
https://medium.com/@snehagiranje05/unveiling-the-magic-understanding-mock-and-magicmock-in-python-ecadf1f1013c